NPS-OR-94-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

OBJECT-ORIENTED SIMULATION PICTURES (OOSPICs) FOR DESIGN AND TESTING

Michael P. Bailey

February 1994

Prepared for:
Naval Postgraduate School
Monterey, CA 93943

NAVAL POSTGRADUATE SCHOOL
MONTEREY, CA 93943-5000

Rear Admiral T. A. Mercer
Superintendent

Harrison Shull
Provost

This report was prepared for the Naval Postgraduate School.

This report was prepared by:

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1994	Technical

4. TITLE AND SUBTITLE
Object-Oriented Simulation Pictures (OOSPics) for Design and Testing

5. FUNDING NUMBERS

6. AUTHOR(S)
Michael P. Bailey

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER
NPS-OR-94-006

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A diagramming technique called Object-Oriented Simulation Pictures (OOSPic) is presented. Using this technique, a simulation designer can show the relationships and interactions between object types. OOSPics also promote extensive bottom-up object testing. Finally, if a complete OOSPic is constructed before coding begins, a reliable model can be constructed directly.

14. SUBJECT TERMS
Object-oriented simulation, Software design

15. NUMBER OF PAGES
14

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

OBJECT-ORIENTED SIMULATION PICTURES (OOSPICs) FOR DESIGN AND TESTING

Michael Bailey

Naval Postgraduate School
Monterey, California U. S. A.
mike@uwhiz.or.nps.navy.mil

ABSTRACT

A diagramming technique called Object-Oriented Simulation Pictures (OOSPic) is presented. Using this technique, a simulation designer can show the relationships and interactions between object types. OOSPics also promote extensive bottom-up object testing. Finally, if a complete OOSPic is constructed before coding begins, a reliable model can be constructed directly.

1 INTRODUCTION

The pursuit of the perfect diagramming technique is a venerable occupation among computer scientists. Among those who specialize in computer simulations, works like Fishman (Fishman 1978) show how flowcharts can be used to design and communicate discrete event models. Languages like SLAM (Pritsker *et al* 1989) or SIGMA (Schruben 1992) were actually designed so that the diagram drawn is the computer simulation. This worked extremely well for languages which concentrated on networks of queues.

Recently, object-oriented simulation methods have become popular, and MODSIM II (MODSIM 1994) has become a tool used by many simulation builders. The product ObjectManager (Object Manager 1994) facilitates model construction in MODSIM, but does not focus on project design. What the designer needs is an easy, standardized way to show relationships between objects, as well as the impacts one object has on another. This work will present a tool useful for designing object-oriented simulations in any object-oriented simulation language which implements process interaction timing. Simula (Birtwistle *et al* 1973), sim++ (Baezner *et al* 1990), SIMSCRIPT (Russell 1983), and Maisie (Bagrodia 1991), all fit this paradigm. We will use MODSIM structures and terminology to facilitate exposition. The reader need not have a background in MODSIM, but must understand the basics of object-oriented simulation, see Cox (Cox 1989), or Taylor (Taylor 1990), for an introduction.

Object-Oriented simulation is not new, and has enjoyed (endured?) a close relationship to other technologies labeled *Artificial Intelligence*. Systems such as Zeigler's (Zeigler 1990) are artifacts of this relationship, and treat the object-oriented simulation designer as a well-trained, state-of-the-art computer scientist. Most simulationists come from the fields of operations research, manufacturing, or engineering, and could use something simpler than Zeigler's DEVS-Scheme.

In this work, we present a simple diagramming technique which Henry Ford would have been proud of:

> "The best ideas are simple."
>
> Henry Ford

OOSPics are at home on chalkboards, backs of envelopes, or in elegant documentation of a large, expensive simulation. The only tools required to produce them are pen, paper, and imagination. While we do have aspirations of automating the OOSPic construction process and adding code generation and testing, the OOSPic is a natural language for simulationists.

We have trained approximately two hundred (200) students in object-oriented design using OOSPics, and have refined and simplified the technique with their help. Diagrams in this article are produced on a computer for publication, but OOSPics are usually drawn by hand.

2 WHY AN OOSPic

When looking at object-oriented model source code, one cannot help but feel as if there are lots of semi-independent entities which relate to

each other in mysterious ways. The Definition Module/Implementation Module breakdown used in MODSIM is beautiful for exposing the workings of a single object, but there is very little to help expose relationships between object types and interactions between objects. Clearly the design process mostly involves these relationships, so we need a method for describing them. We call this method OOSPic, short for Object-Oriented Simulation Pictures.

These diagrams facilitate

- COMMUNICATING REQUIREMENTS;
- DESIGNING MODELS;
- TESTING MODELS;
- DOCUMENTING MODELS;
- LEARNING ABOUT OOS.

One complete OOSPic is composed of three different kinds of diagrams:

1. Object Lay-Out: shows the relationships between object type definitions, as well as relaying information about how the objects will act.
2. Transition/Action Diagram: shows the flow of a single method, and emphasizes the external actions of the object.
3. System Drop-Through: simple flowchart showing how the simulation is executed.

The object lay-out shows which objects inherit properties from other objects, set memberships, and different types of ownership. Before we can discuss these relationships, we need to establish that a single object in an OOSPic is shown as a tagged, four-layered box, shown in Figure 1.

1. The top layer states the object's type name.
2. The second layer lists all of the fields of the object.
3. The third layer lists all of the ASK (non-time-consuming) METHODS of the object.
4. The fourth layer lists all of the TELL (time-consuming) METHODS of the object.

The tag gives the object another name, called its local name. We'll see the importance of tagging when we discuss object ownership.

It's important to state that we are not suggesting that all of this information be listed every time the object box is used in the OOSPic. For example, there

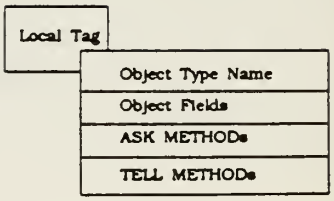

Figure 1: An Object Box

may be times when an object box is used to identify a single ASK METHOD of the object. In this case, we simply omit the other lists. However, we should keep the separating bars in the box so that positional relationships are maintained.

2.1 Motivating Example - Project 1 of Introduction to System Simulation: Remotely Controlled Vehicle

The following is a description of a project undertaken in an introductory system simulation course which uses MODSIM. We will use this project to motivate our discussion of the OOSPic. The project assignment is stated as follows:

Design an object which represents a vehicle, person, animal, storm system, etc., which is called DirectedMovingObj. This object must be controlled by inputting directions (speed, waypoints) from the console. All of the user interactions and object actions should be recorded in an output file, along with entries and exits from all methods and procedures. The following are the steps involved:

1. Design an object called MovingObj which maintains its own position and velocity, and which takes directions. This object should start at (0,0) and have an initial velocity of 0. The definition module for MovingObj is shown in Figure 2.
2. Inherit the MovingObj into DirectedMovingObj, which is a MovingObj which gets directions from the user console by asking questions. The object should OUTPUT its position and the current simulation time at the end of each move. Hint: This object should WAIT FOR itself to MoveTo in the TELL METHOD DoGuidedTour. Figure 3 shows the definition for DirectedMovingObj.
3. In a PROCEDURE create a variable number of objects of type DirectedMovingObj and place them in an object of type QueueObj. TELL each to DoGuidedTour.

```
DEFINITION MODULE Moving;
{
        Mike Bailey
}
TYPE
XYRecType = RECORD
    X : REAL;
    Y : REAL;
END RECORD;

MovingObj = OBJECT
    Position : XYRecType;
    Velocity : REAL;
    ASK METHOD ObjInit;
    TELL METHOD MoveTo(IN XY : XYRecType);
    ASK METHOD ChangeVelocity(IN Vel : REAL);
END OBJECT;
END MODULE.
```

Figure 2: Definition module for the MovingObj, found in file DMoving.mod

```
DEFINITION MODULE Direct;
{
        Mike Bailey
}
FROM Moving IMPORT MovingObj;

TYPE
DirectedMovingObj = OBJECT(MovingObj)
    TELL METHOD DoGuidedTour;
    {DoGuidedTour manages the movement of
    the object by interacting with the
    user to get directions.}
END OBJECT;
```

Figure 3: Definition module for DirectedMovingObj, found in file DDirect.mod

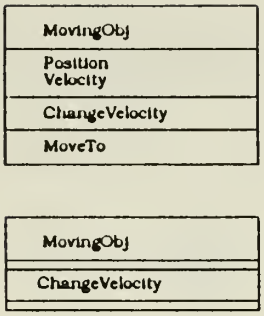

Figure 4: Object Boxes for the MovingObj, Full and Abbreviated.

Figure 5: Arrow Symbols used in an Object Lay-Out. From left to right, they are inheritance, permanent ownership, temporary ownership, and membership.

Thus, suppose that we had an object MovingObj which had an object box as shown in Figure 4. If we wanted to use an object box which just highlighted the ASK METHOD Change Velocity, we could use the abbreviated box shown.

3 OBJECT LAY-OUT

An object lay-out should tell us everything required to define an object. It also includes several aspects which will help the designer relay his intentions about how an object may be used.

An object lay-out contains an object box for every object in the model. Special arrows connect the boxes to show

- inheritance;
- permanent ownership.
- temporary ownership;
- membership;

Our DirectedMovingObj shows its inheritance relationship to MovingObj in Figure 6. In addition, it shows the membership relationship as each DirectedMovingObj is a member of the QueueOfMovers.

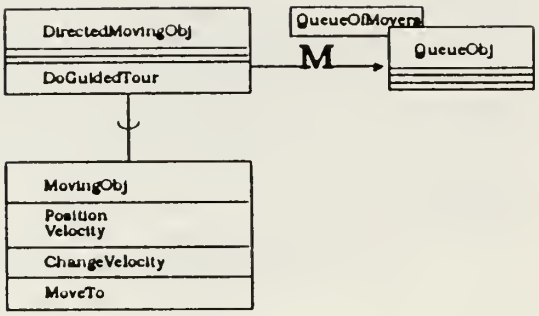

Figure 6: Object Lay-Out for the MovingObj Project.

Some interesting notes about the object lay-out of DirectedMovingObj:

1. This is the lay-out of a subset of the key components in the simulation model – all of these diagrams are useful in building and documenting pieces of models.
2. If DirectedMovingObj included an OVERRIDE of a METHOD of MovingObj, we would see this by observing the same METHOD name in both MovingObj and DirectedMovingObj.
3. QueueOfMovers is the local name of the QueueObj object which contains DirectedMovingObj instances. The object box tag tells us this.
4. The QueueObj object box is empty, no details are required. QueueObj is a standard MODSIM object type and is well documented in (MODSIM 1994).

To see the ownership features of an object lay-out, let's suppose that we have an object called a VehicleObj, which inherits all of the properties of the MovingObj, and which has a permanent Engine and which may carry Cargo. The object lay-out for the engine is shown in Figure 7.

Note the two different kinds of arrow relating the Engine to the VehicleObj and the Cargo to the VehicleObj. From this diagram we can expect many different LoadObj's to be attached to a VehicleObj, however the VehicleObj has a single permanent Engine during the simulation. Also notice the use of the tags. This identifies the PowerPlantObj with the Engine field of the VehicleObj. The VehicleObj designer may even decline to list Engine as a field of VehicleObj because of the tag on the PowerPlantObj.

4 TRANSITION/ACTION DIAGRAMS

Each important METHOD of each object should have its own transition/action diagram. This diagram is

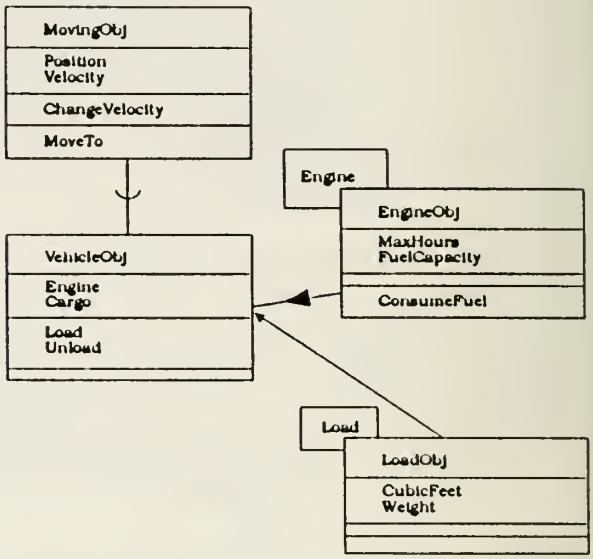

Figure 7: Object Lay-Out Showing Permanent and Temporary Ownership. The Engine is a permanent component of the VehicleObj, but the Cargo may be swapped in and out several times during the simulation.

closely related to the well-known flowcharts first used in the 1950's, but has been tailored to object-oriented simulation use.

4.1 Transitions

In each METHOD of an object, the object may be thought of as rolling through a set of states. These states are of two distinct types:

1. PERSISTENT STATES: The object is suspended and simulation time is elapsing.
2. TRANSIENT STATES: The object is doing one or both of the following:
   (a) developing some important data product;
   (b) interacting with other objects.

In these cases, the simulation clock is frozen while the object is in the transient state. See Figure 8. We use boxes to indicate states, thick walls for persistent states and thin walls for transient states.

The METHOD flows from one state to another according to simple arrows. In order to show logical flow, we need two relics of the traditional flow chart, the decision and the loop. Decisions are indicated by diamonds, while loops flow with the arrows. See Figure 9.

Figure 8: State Boxes for Transient (thin walls) and Persistent (thick walls) States.

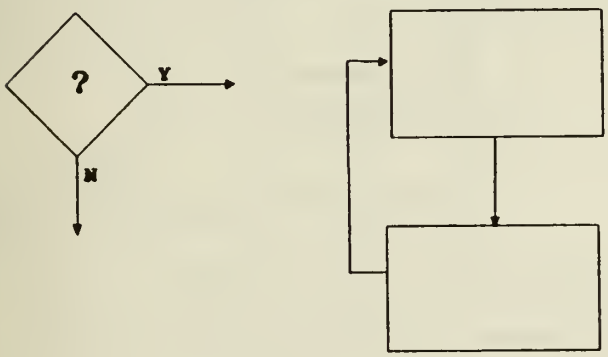

Figure 9: Decisions and Loops.

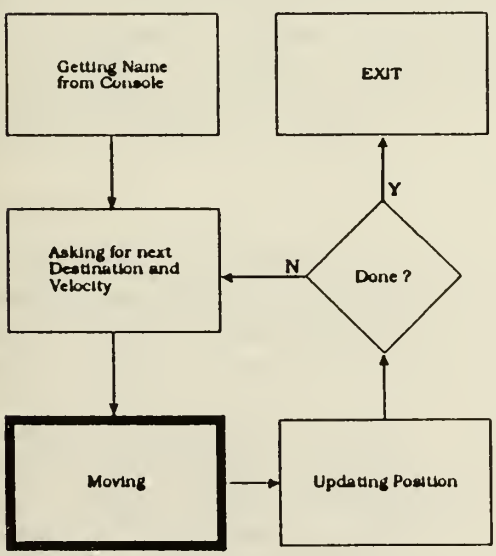

Figure 10: Transition/Action Diagram for DoGuidedTour. Only the transitions are shown.

These symbols are all that we need to describe the transitions that the object takes on. The TELL METHOD DoGuidedTour of the DirectedMovingObj has the transitions as shown in Figure 10. Note the use of the -ing suffix in all of the state descriptions. This encourages us to anthropomorphize the object, and to account for all of its actions.

When designing an object METHOD, the designer can simply scribble out the states as they come to his mind. However, when preparing more formal transition/action diagrams, the separation of states can be important. We suggest that the separation be as follows:

1. PERSISTENT STATES: one state identified per WAIT in the implementation code.
2. TRANSIENT STATES: one state identified per important data product, and one state per important action.

4.2 Actions

Actions are all those things which an object does that involve other METHODS or other objects. These actions include:

1. starting other METHODS;
2. querying other object's fields or invoking ASK METHODS which return values;
3. starting an INHERITED method;
4. delivering data to another object;
5. WAITing for another object to finish a TELL METHOD.

The sequence in Figure 11 shows the symbols for each of these actions.

1. Starting a TELL METHOD: a simple arrow from a transient state box to an object box with its TELL METHOD listed.
2. Getting data: looping arrow from a transient state box to an object box, the information provided is shown in the object box by listing fields or ASK METHODS which return values.
3. Delivering data: straight arrow from transient state box to an object box. The object box shows the ASK METHOD which receives the data.
4. Inherited METHOD: thick arrow pointing into a persistent or transient state box from an object box. The object box shows the METHOD inherited, the type of the state box depends on

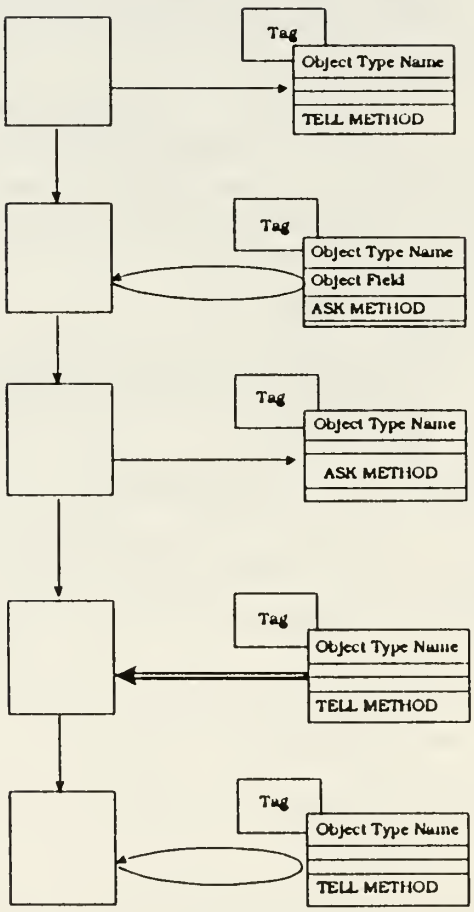

Figure 11: Actions. From top to bottom, they are start a TELL METHOD; Get data from a field or by invoking an ASK METHOD; Deliver data to another object; Start an inherited METHOD; WAIT FOR a TELL METHOD

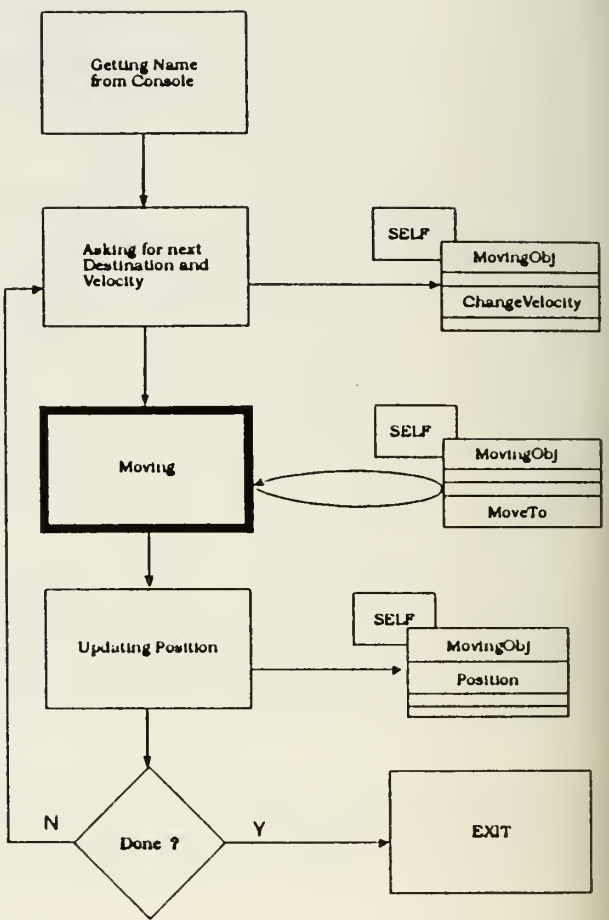

Figure 12: Full Transition/Action Diagram for DoGuidedTour.

whether the inherited METHOD is an ASK or TELL.

5. WAIT FOR: looping arrow from persistent state box to an object box. The object box lists the TELL METHOD which is WAITed FOR.

The full transition/action diagram for DoGuidedTour is shown in Figure 12. Note that the inherited METHODs involved are labeled as actions taken with SELF as a MovingObj. The SELF label always tips off the use of an inherited METHOD. Hence, the DirectedMovingObj will deliver data to its own ASK METHOD ChangeVelocity, it will WAIT FOR itself to execute the TELL METHOD MoveTo, and it will change its own Position field.

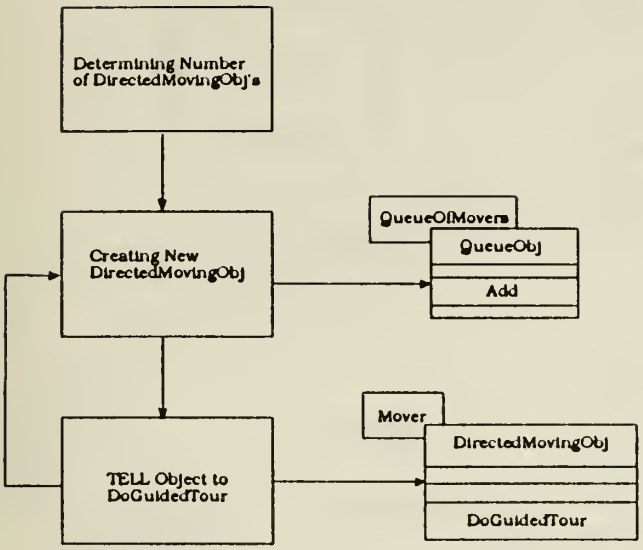

Figure 13: System Drop-Through.

5 SYSTEM DROP-THROUGH

Simulation models run inside simulation executives. This executive subprogram is not a METHOD of an object, it's simply a PROCEDURE which initializes the system, does replications, collects data, and performs output analysis procedures. Often a simulation is reused to do several different analyses by changing only this process.

The diagram we use to design and communicate the simulation executive is called the system drop-through. It consists of a single transition/action diagram with no persistent states. It may be decomposed into modular procedures, but it always has a linear flow. Below we see the simple system drop-through for the project.

6 TESTING

Testing any computer program relies mostly on common sense and careful work. Testing an object-oriented simulation is often a difficult concept because of the object's autonomy and flexibility. However, these were the same reasons we used to justify constructing the OOSPic. As it turns out, our pictures facilitate testing our objects and our system. We should pursue testing in the nested frameworks shown in Figure 14.

Manufacturing technology of durable goods like automobiles was revolutionized by emphasizing testability in the *design* of the product. A modern car designer includes test equipment in the car's design, and specifies the testing the car will undergo during

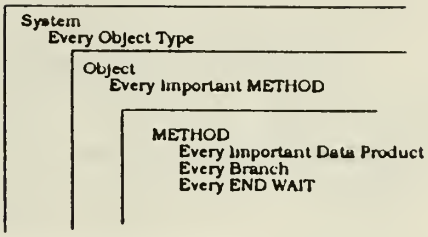

Figure 14: Nested Testing.

manufacture. We should do the same in our software design. Planning tests must be done during the design phase of the project. We start from the inside and work our way out.

6.1 METHOD Tests

Action/transition diagrams show the decomposition of METHODS into states where important data products are developed, decisions are made, and simulation time elapses. To test the performance of a METHOD, we simply need to verify that the developed data products are correct, the decisions made are the right ones, and that the timing of the WAITs is correct.

Hence, transition out of a state is an obvious point for checking the state's results. These results are:

- values of data developed or attained;
- results of decisions;
- the simulation clock time after a persistent state.

Each METHOD should be exercised in every branch and for every important, foreseeable situation.

6.2 Object Tests

Looking at an object lay-out, we find everything in an object that should be tested – these are the METHODs of the object. The suggested methodology is to construct a testing MAIN MODULE for each object type. This process should start with the objects which are base types, those that do not inherit any other object's properties, and which are not owned by another object. One instance of the object should be created. The object's METHODs should all be exercised and all of the output should be collected in a file. This file's contents should be checked for correct responses from the object, then should be renamed and kept.

Hence, at the end of this process, we have files:

1. "M" + object name + ".mod" – the MAIN used to test the object.

2. object name + ".test" (or ".tst") – output from execution of the test program.

These test programs and their results should be archived by the developer. When the object is changed or inherited, the test program can be recompiled and rerun to check for consistency with older versions.

6.3 System Tests

Finally, the system drop-through diagram can be used to generate tests of the entire model. Every state in the system drop-through is a breakpoint where correct performance can be tested. Let's look at the OOSPic for the Mover project. We wish to test the METHOD, OBJECTS, and SYSTEM for this project.

Testing the system with the DirectedMovingObj objects can be partially done as follows. Inside the TELL METHOD DoGuidedTour, the following information should be collected as indicated in the transition/action diagram:

1. the Name assigned to the object;
2. the destination and velocity the user requests;
3. the simulation clock time at the end of the WAIT;
4. the object's new position.

Figure 15 shows the transition/action diagram for DoGuidedTour annotated for testing. When planning testing using a hand-drawn OOSPic, we suggest that the designer photocopy the transition/action diagram and use colored pens to make the testing annotations.

Testing the DirectedMovingObj should not be undertaken until the MovingObj is tested. Once that is accomplished and the files MMovingObj.mod and MovingObj.tst are safely tucked away, we can create MDirectedMovingObj.mod. The contents of this module can be seen in Figure 16.

7 OOSPics for ADVANCED MODSIM

In this final section, we address some more advanced MODSIM capabilities which can also be diagrammed using OOSPics. These are:

1. Interrupts of TELL METHODS;
2. TriggerObjs: used to synchronize object activities by using an ASK METHOD Fire;
3. ResourceObjs: manage a pool of resources.

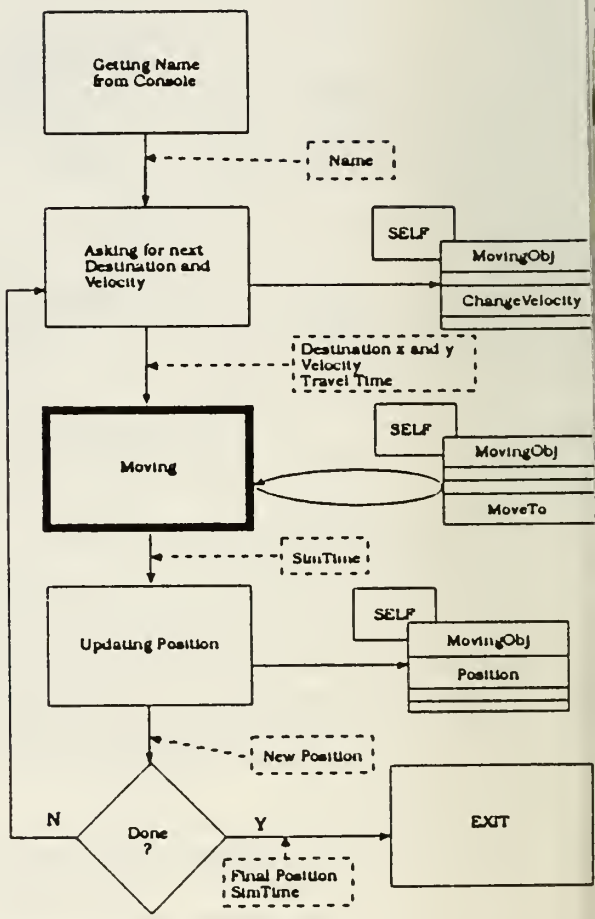

Figure 15: Testing-Annotated Transition/Action D agram.

```
MAIN MODULE DirectedMovingObj;
FROM Direct IMPORT DirectMovingObj;

VAR
      Mover: DirectedMovingObj;

BEGIN
   NEW(Mover);
   TELL Mover TO DoGuidedTour;
END MODULE.
```

Figure 16: Definition module for the MovingOb found in file DMoving.mod

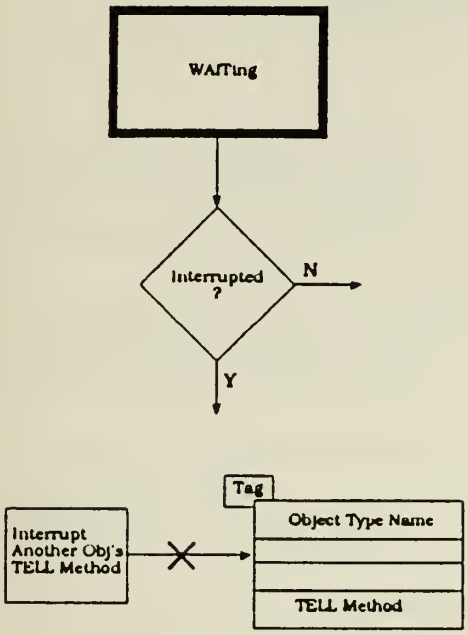

Figure 17: Action Diagram for an Interrupt.

7.1 Interrupt

Any persistent state is associated with a WAIT in the MODSIM code, so the exit from the persistent state can be caused by the WAIT ending successfully, or by the WAIT being interrupted. The latter may be handled in the OOSPic transition/action diagram using a simple decision.

To show a METHOD causing an interruption of another object's TELL METHOD, we need a new action symbol, seen in Figure 17 as the arrow with the "X" on it. The TELL METHOD interrupted is shown in the object box.

7.2 TriggerObj

A trigger object can impact another object by stopping the flow of a TELL METHOD until the trigger object executes its Fire Method. Hence, the appropriate symbol is the WAIT FOR. Distinguishing the WAIT FOR trigger from the generic WAIT FOR only requires a look at the object box, where an ASK METHOD Fire is shown.

7.3 ResourceObj

WAITing for a resource is like WAITing for a trigger, except that resource WAITing can be specialized using timed WAIT FORs, or prioritized WAIT FORs. In either case, simple annotation is all that is required to show these. The object box must show the Give METHOD as the METHOD we WAIT FOR. In the case of the TriggerObj and the ResourceObj, the

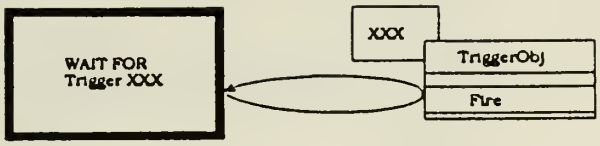

Figure 18: Action Diagram for using a TriggerObj.

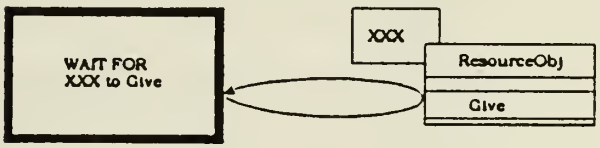

Figure 19: Action Diagram for using a ResourceObj.

reader knows something funny is going on because the diagram shows a persistent state WAIT FORing an ASK METHOD.

8 CONCLUSION

In this work, we have presented a way to design and document object-oriented simulation models using diagrams. We have focused on one particular OOS language, MODSIM, only so much as we have incorporated the essential capabilities and vernacular of this language. The diagramming paradigm, OOSPic, allows us to communicate about the structure of objects and their interactions during simulation. It is foreseen that some tool similar to ObjectManager will eventually incorporate and enhance this diagramming technique.

REFERENCES

Baezner, D., G. Lomow, and B. Unger. 1990. Sim++: The Transition to Distributed Simulation. *Distributed Simulation, Vol. 22.* San Diego: Society of Computer Simulation. p. 211-218.

Bagrodia, R. 1991. Interative Design of Efficient Simulations using Maisie. *Proceedings of the Winter Simulation Conference, B. Nelson, D. Kelton, and G. Clark, Eds.* p.243-47.

Bertwistle, G., O.-J. Dahl, B. Myhrhaug, and K. Nygaard. 1973. *SIMULA BEGIN.* Phiadelphia: Auerbach.

Booch, G. 1990. *Object Oriented Design with Applications.* Redwood City, California: Benjamin/Cummings.

Cox, B. J. 1989. *Object Oriented Programming: An Evolutionary Approach.* New York: Addison-Wesley.

Fishman, George S. 1978. *Principles of Discrete Event Simulations.* New York: Wiley & Sons.

MODSIM II Programming Language Reference Manual, Version 1.9 1994. LaJolla, California: CACI.

Object Manger Reference Manual. 1994. LaJolla, California: CACI.

Pritsker, A. A., C. E. Sigal, and R. D. Hammesfahr. 1989. *SLAM II Network Models for Decision Support.* Englewood Cliffs, New Jersey: Prentice-Hall.

Russell, E. C. 1983. *Building Simulation Models in SIMSCRIPT II.5.* LaJolla, California: CACI.

Schruben, L. 1992. *SIGMA: A Graphical Simulation System.* San Francisco: Scientific Press.

Taylor, D. A. 1990. *Object-Oriented Technology: A Manager's Guide.* San Francisco: SERVIO.

Weiner, R. S. and L. J. Pinson. 1988. *An Introduction to Object-Oriented Programmming and C++.* New York: Addison-Wesley.

Zeigler, B. 1990. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* San Diego: Academic Press.

INITIAL DISTRIBUTION LIST

1. Research Office (Code 08) 1
   Naval Postgraduate School
   Monterey, CA 93943-5000

2. Dudley Knox Library (Code 52) 2
   Naval Postgraduate School
   Monterey, CA 93943-5002

3. Defense Technical Information Center 2
   Cameron Station
   Alexandria, VA 22314

4. Department of Operations Research (Code OR) 1
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Prof. Michael P. Bailey (Code OR/Ba) 50
   Naval Postgraduate School
   Monterey, CA 93943-5000